Proposta de Paralelismo Semi-Automático de Pipelines Lineares em C++

Renato B. Hoffmann, Dalvan Griebler

[1] Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brasil

renato.hoffmann@edu.pucrs.br,dalvan.griebler@pucrs.br

Resumo. *Atingir alto desempenho, portabilidade e produtividade continua sendo um desafio fundamental na comunidade de computação de alto desempenho. Entretanto, a paralelização automática de programas nunca foi amplamente adotada devido à sua inconsistência. Assim, a tarefa onerosa da programação paralela permanece sob a responsabilidade direta dos programadores. Esse trabalho propõe-se estudar técnicas de paralelismo com compiladores.*

Com base em análise da literatura, identificamos que compiladores totalmente automáticos de paralelização frequentemente produzem resultados inconsistentes [Harel and et al. 2019]. Portanto, esse trabalho segue uma abordagem semiautomática, que combina análise de compilador com a intervenção do programador por meio de anotações. O foco é em aplicações de processamento de fluxos modeladas como pipelines lineares, um domínio amplamente utilizado, mas pouco explorado na paralelização por compiladores [Tournavitis and Franke 2010]. Nosso objetivo é atingir o mínimo esforço de programação com uma paralelização confiável e eficiente. Entretanto, esse é um objetivo muito amplo. O primeiro passo consiste na criação da representação de código que permitirá realizar as análises e transformações futuras para habilitar o paralelismo.

Construindo Grafo de Dependências a partir da AST. O primeiro passo na paralelização por compilador é a construção do Grafo de Dependência do Programa (PDG), que serve como base para a análise de dependências de dados e particionamento de estágios. A representação de código adotada é a Árvore de Sintaxe Abstrata (AST) do Clang. Durante a construção do PDG, todas as variáveis devem ser visitadas. Variáveis na AST do Clang aparecem como declarações ou referências.

Para construir o PDG, é necessário visitar quatro tipos principais de nós da AST: `DeclRefExpr`, `BinaryOperator`, `UnaryOperator` e `VarDecl`.

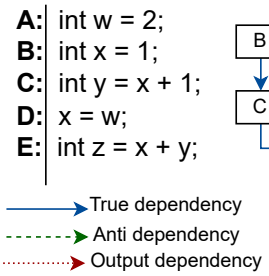
VarDecl: A visita aos nós de declaração de variáveis ajuda a detectar variáveis de escopo local e as variáveis usadas para construí-las. Embora idealmente todas as especializações seriam consideradas, bastou tratar os nós `VarDecl` para cobrir também os outros tipos especiais devido à estrutura da AST e à estrutura orientada a objetos (por exemplo, `FieldDecl` é uma subclasse de `VarDecl`).

BinaryOperator: A visita aos nós de operadores binários permite rastrear modificações em variáveis em cada nova atribuição. Ao analisar recursivamente o lado direito da expressão (RHS), pode-se identificar todas as variáveis envolvidas no cálculo da atribuição. Também pode ser necessário analisar recursivamente o lado esquerdo (LHS) para aritmética de ponteiros, como em `ptr + var = anotherVar`.

UnaryOperator: Detecta modificações auto-referenciadas em variáveis, como `var++`.

DeclRefExpr: Identifica todas as variáveis referenciadas dentro do escopo de uma região paralela. No Clang, um `DeclRefExpr` representa uma referência a uma variável em um determinado ponto do ciclo de vida do programa. Essa referência pode ser posteriormente substituída por um registrador, uma operação de memória, uma constante em tempo de compilação ou até eliminada por otimização de código morto, mas isso não impacta o nível de análise atual.

Para representar o PDG, foi utilizado um `unordered_map` que mapeia nós `DeclRefExpr *` para um *unordered_set* de `DeclRefExpr *`, formando uma estrutura de grafo. A Figura 1 ilustra um grafo de dependência gerado por meio da travessia da AST do Clang, juntamente com seu código C correspondente. O conceito de `Ref` denota referências a nós da AST do Clang, representando pontos de execução do programa. Posteriormente, esse grafo de dependência servirá como base para a detecção de Componentes Fortemente Conectados (SCC) e particionamento de estágios.



True Dependencies	
Base Ref	**Depends On**
Ref(w→**A**)	-
Ref(x→**B**)	-
Ref(y→**C**)	[Ref(x→**B**)]
Ref(x→**D**)	[Ref(w→**A**)]
Ref(z→**E**)	[Ref(x→**D**), Ref(y→**C**)]

Figura 1. Exemplo de grafo de dependência. A coluna Base Ref é um *DeclRefExpr e a coluna Depends On é um `unordered_set<DeclRefExpr*>`.**

Durante a criação do PDG, também foi possível coletar informações sobre todas as variáveis do escopo da região paralela: (1) variáveis locais, (2) constantes externas e (3) variáveis externas que são modificadas. A combinação dessas informações com a análise do PDG permite identificar as transformações necessárias para o paralelismo. Variáveis locais não requerem tratamento especial, constantes externas podem ser copiadas, e variáveis externas sujeitas a atribuições contribuem para dependências. As transformações para resolver as dependências são: privatização, redução, resolução de indução, utilização de atômicos, barreiras e outros algoritmos de sincronização, e introdução de exclusão mútua por software [Bhosale and et al. 2022]. [1]

Referências

Bhosale, A. and et al. (2022). Automatic and interactive program parallelization using the cetus source to source compiler infrastructure v2.0. *Electronics*, 11:1–22.

Harel, R. and et al. (2019). Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: Analysis, pitfalls, enhancement and potential. *International Journal of Parallel Programming*, 48:1–31.

Tournavitis, G. and Franke, B. (2010). Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *PACT*, page 377–388, Vienna, Austria. Association for Computing Machinery.

[1]Esta pesquisa foi financiada com o apoio de FAPERGS 09/2023 PqG (Nº 24/2551-0001400-4) e CNPq Research Program (Nº306511/2021-5)